Unit testing C# with NUnit and .NET Core

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

Creating the source project

Open a shell window. Create a directory called *unit-testing-using-nunit* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution file for the class library and the test project. Next, create a *PrimeService* directory. The following outline shows the directory and file structure thus far:
Copy

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
```

Make *PrimeService* the current directory and run `dotnet new classlib` to create the source project. Rename *Class1.cs* to *PrimeService.cs*. To use test-driven development (TDD), you create a failing implementation of the `PrimeService` class:
C#Copy

```
using System;

namespacePrime.Services
{
publicclassPrimeService
    {
publicboolIsPrime(int candidate)
        {
thrownew NotImplementedException("Please create a test first");
        }
    }
}
```

Change the directory back to the *unit-testing-using-nunit* directory. Run `dotnet sln add PrimeService/PrimeService.csproj` to add the class library project to the solution.

Install the NUnit project template

The NUnit test project templates need to be installed before creating a test project. This only needs to be done once on each developer machine where you'll create new NUnit projects. Run `dotnet new -i NUnit3.DotNetNew.Template` to install the NUnit templates.
Copy

```
dotnet new -i NUnit3.DotNetNew.Template
```

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:
Copy

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using `dotnet new nunit`. The dotnet new command creates a test project that uses NUnit as the test library. The generated template configures the test runner in the *PrimeServiceTests.csproj* file:
XMLCopy

```
<ItemGroup>
<PackageReference Include="Microsoft.NET.Test.Sdk"Version="15.5.0" />
<PackageReference Include="NUnit"Version="3.9.0" />
<PackageReference Include="NUnit3TestAdapter"Version="3.9.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added the Microsoft test SDK, the NUnit test framework, and the NUnit test adapter. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:
Copy

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

You can see the entire file in the samples repository on GitHub.

The following outline shows the final solution layout:
Copy

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.csproj
```

Execute `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.csproj` in the *unit-testing-using-dotnet-test* directory.

Creating the first test

The TDD approach calls for writing one failing test, making it pass, then repeating the process. Remove *UnitTest1.cs* from the *PrimeService.Tests* directory and create a new C# file named *PrimeService_IsPrimeShould.cs* with the following content:
C#Copy

```
using NUnit.Framework;
using Prime.Services;

namespacePrime.UnitTests.Services
{
    [TestFixture]
publicclassPrimeService_IsPrimeShould
    {
privatereadonly PrimeService _primeService;

publicPrimeService_IsPrimeShould()
        {
            _primeService = new PrimeService();
        }

        [Test]
publicvoidReturnFalseGivenValueOf1()
        {
var result = _primeService.IsPrime(1);

            Assert.IsFalse(result, "1 should not be prime");
        }
    }
```

```
}
```

The `[TestFixture]` attribute denotes a class that contains unit tests.
The `[Test]` attribute indicates a method is a test method.

Save this file and execute `dotnet test` to build the tests and the class library and then run the tests. The NUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:
C#Copy

```
publicboolIsPrime(int candidate)
{
if (candidate == 1)
    {
returnfalse;
    }
thrownew NotImplementedException("Please create a test first");
}
```

In the *unit-testing-using-nunit* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add new tests with the `[Test]` attribute, but that quickly becomes tedious. There are other NUnit attributes that enable you to write a suite of similar tests. A `[TestCase]` attribute is used to create a suite of tests that execute the same code but have different input arguments. You can use the `[TestCase]` attribute to specify values for those inputs.

Instead of creating new tests, apply this attribute to create a single data driven test. The data driven test is a method that tests several values less than two, which is the lowest prime number:
C#Copy

```
[TestCase(-1)]
[TestCase(0)]
[TestCase(1)]
publicvoidReturnFalseGivenValuesLessThan2(intvalue)
{
var result = _primeService.IsPrime(value);

    Assert.IsFalse(result, $"{value} should not be prime");
}
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the method:
C#Copy

```
if (candidate < 2)
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.